SparkSql

1、Spark SQL

Spark SQL是Spark的五大核心模块之一，用于在Spark平台之上处理结构化数据，利用Spark SQL可以构建大数据平台上的数据仓库

2、Spark SQL的入口：SQLContext

1）SQLContext：是SparkSQL模块SQL编程的入口，主要提供SparkSQL的基本SQL语言的一种处理方式，主要应用于除和hive集成以外的其他应用场景中

2）HiveContext：是SparkSQL和hive集成项目需要的SQL编程入口，当SparkSQL应用需要读取hive元数据、hive的相关功能（SQL不提供，但hive提供，比如窗口分析函数）等，就必须使用HiveContext

3、SparkSQL的read和write编程模型

1）read相关函数（DataFrameReader）

eg：val df = sqlContext.read.#

相关API：

1. Format：给定数据读取的方式（指定数据使用哪个类进行读取，可以给缩写、包名或者全类名称），常用的：json（默认）、parquet
2. schema：给定schema信息，如果不给定，会自动的进行
3. option：给定读取数据的时候需要的参数信息
4. load:加载数据形成DataFrame（一般用于除了hive、jdbc以外的其他数据源）
5. jdbc：加载jdbc的数据形成Dataframe
6. Table：根据给定的表名称加载数据形成Dataframe，给定的表名称可以是临时表名称也可以是hive中的表名称

2）write相关函数（DataFrameWriter）

eg：DataFrame.write.#

相关API：

1. format：给定数据存储的方式(指定数据使用哪个类进行存储，可以给缩写、包名或者全类名称)。常用的：json（默认）、parquet
2. mode：给定当输出的文件夹、表存在的情况下，进行什么操作

Case “overwrite” =>SaveMode.Overwrite

表示当存在的时候，进行覆盖操作；也就是直接把文件夹/表删除(所有数据)，然后重新的构建

Case “append” => SaveMode.Append

表示追加数据，直接添加到文件夹、表中

Case “ignore” => SavaMode.Ignore

表示如果存在，那么如果存在，那么不进行数据输出操作

Case “error” => SaveMode.ErroeIfExists

默认操作，如果存在报错

1. option：给定数据输出过程中需要涉及到的参数
2. partitionBy：当支持分区存储的时候，给定分区字段，可以是多个
3. Save：触发数据保存操作,一般用于除了jdbc和hive之外的应用场景
4. insertInto：将Dataframe的数据插入到一张已经的hive表中（不能是临时表）
5. saveAsTable：将Dataframe的数据保存到一张hive表，表可以不存在
6. Jdbc：将Dataframe的数据写出到关系型数据库中，如果需要实现Insert or Update操作的话(当记录/数据主键存在的时候，进行更新数据操作；当不存在，进行数据插入操作)，一般该API不适合，选择使用foreachPartition API自定义数据输出代码

3）jdbc的三种API

(1) jdbc(url：String , table：String，properties：Properties)

给定URL、table名称以及连接参数获取数据库的数据形成Dataframe，但形成DataFrame只有一个分区

1. Jdbc(

url：String, //数据库连接参数

table：String, //读取数据的数据库表名称

columnName：String, // 进行分区的字段 -》必须是整型的

lowerBound：Long, // 进行分区范围计算的下界

upperBound：Long, // 进行分区范围计算的上界

numPartitions：Int // 给定的分区数，也就是最终形成的Dataframe中的分区数

connectitionProperties：Properties

)

要求进行分区的字段数据类型必须是数值型的

最终形成的DataFrame中的分区数量即参数numPartitions；每个分区处理的数据范围是：val step = ((upperBound - lowerBound) / numPartitions).toLong

var currentIndex = lowerBound + step ==> 得到第一个分区(负无穷大，currentIndex)

var preIndex = currentIndex

while(分区数小于numPartitions-1) {

currentIndex = currentIndex + step ==> 得到分区范围是 [preIndex, currentIndex)

preIndex = currentIndex

}

最后一个分区的范围是 ==> [currentIndex, 正无穷大)

3）jdbc(

url：String,

table：String,

predicates：Array[String], // 给定分区条件(where条件，一般选择用主键来分区) Array[String]( "deptid <=20","and deptid >20 and deptid <=40","and deptid > 40 ")

connectionProperties：Properties

)

最终形成的DataFrame中的分区数量是参数predicates里面的条件数量；而且每个分区中的数据均满足predicates里面对应的where数据过滤条件

1. 案列：读取HDFS之上的json文件形成DataFrame，然后在写sql处理

1）读取文件形成dataframe

\*\*注意：json文件数据要求，每一行是一个json对象

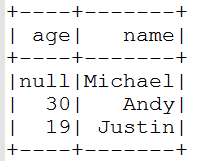
val df1 = sqlContext.read.json("/spark/data/resources/people.json")

val df2= sqlContext.read.format("json").load("file:///opt/cdh-5.3.6/spark/examples/src/main/resources/people.json")

df1: org.apache.spark.sql.DataFrame = [age: bigint, name: string]

df1读取HDFS上文件，df2读取linux上的文件

df1.show()



2）将Dataframe形成临时表

(1) 注册的临时表只有sqlcontext内部维护的一个临时表名称和Dataframe之间的映射

(2) 临时表没有databases(数据库)的概念

(3) 注册的临时表的时候，表的名称中不能有符号“.”

3）具体的业务开发（写SQL语句）

sqlContext.sql("select name from people1").show

sqlContext.sql("select name,age+1 from people1").show

sqlContext.sql("select \* from people1 where age > 21").show

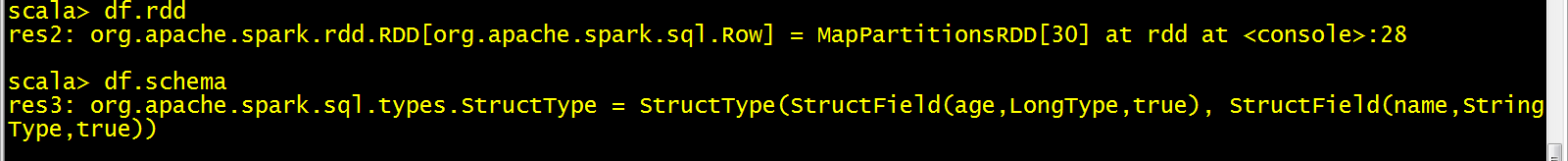
sqlContext.sql("select age,count(\*) from people1 group by age").show

5、DataFrame

1）Dataframe底层是以RDD为基础的分布式数据集，和RDD的主要区别的是：RDD中没有schema信息，而DataFrame中数据每一行都包含schema

Dataframe由两部分组成RDD[Row] + schema

1. RDD：DataFrame中的数据是：是以Row为数据进行存储的
2. schema：Dataframe中的数据对应的schema信息，包含字段名称，字段数据类型



1. DataFrame的实质性上是一个逻辑计划(类似于rdd的懒加载)，只有到数据要读取的时候，才会将逻辑计划进行分析和优化，并最终转化为RDD
2. SparkSQL的开发流程
3. 读取数据形成DataFrame

① 读取hive表中的数据

val df = sqlContext.sql(“select \* from hive数据库**.**表名”)

val df = sqlContext.table(“hive数据库.表名”)

②读取非hive表数据

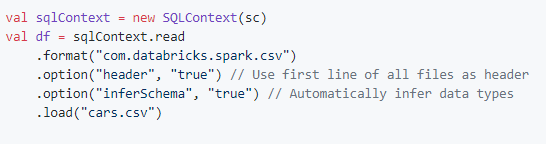
A.读取内置数据源mysql json parquet

val df = sqlContext.read

.jdbc（MySQL）.json（json）.parquet（parquet）

B.读取外部数据源 HBase csv

val df = sqlContext.read.option.load



(2) DataFrame的数据处理

① 使用SQL或者hql 语句

先注册成临时表registerTempTable，然后基于临时表使用 SQLContext.sql进行处理

② DSL语法处理

DSL可以直接调用Dataframe的api 进行处理：select()等

(3) 结果输出

① 转成RDD，然后对RDD进行数据输出 ------>>> HDFS

② saveAsTable存入hive表中 ------>>> hive

③ 调用DataFrame的api 进行输出，使用sqlContext的write编程模型保存数据（MySQL、json、xml、csv）

df.write.jdbc df.write.format

6、RDD和DataFrame的相互转化

1）RDD ----->>>> DataFrame

(1) 利用反射推断scheme

(2) 编程指定schema信息

case class Employ(empid:Int,empname:String,job:String,mgid:Int,hiredate:String,

salary:Double,bonus:Double,deptid:Int)

//1.分别读取dept.txt 和 emp.txt

val deptRdd = sc.textFile("C:\\Users\\Administrator\\Desktop\\Code\\data\\dept.txt")

val empRdd = sc.textFile("C:\\Users\\Administrator\\Desktop\\Code\\data\\emp.txt")

//2.分别使用不同的方式将这2个RDD转成DataFrame

//2.1 使用 利用反射自动推断schema信息 要求rdd的每一个元素为case calss

val empDF = empRdd.map(\_.split("\t"))

.map(arr =>{

val empid = if (arr(0).nonEmpty) arr(0).toInt else 0

val empname = arr(1)

val job = arr(2)

val mgid = if(arr(3).nonEmpty) arr(3).toInt else 0

val hiredate = arr(4)

val salary = if(arr(5).nonEmpty) arr(5).toDouble else 0.0

val bonus = if(arr(6).nonEmpty) arr(6).toDouble else 0.0

val deptid = if(arr(7).nonEmpty) arr(7).toInt else 0

Employ(empid,empname,job,mgid,hiredate,salary,bonus,deptid)

}).toDF

empDF.registerTempTable("emp")

empDF.show

//2.2 通过编程指定schema方式将deptRdd转成DataFarme

要求：rdd每一行是Row 指定schema

val schema = StructType(Array(

StructField("deptid",IntegerType,true),

StructField("deptname",StringType,true),

StructField("addr",StringType,true)

))

val deptRow = deptRdd.map(\_.split("\t"))

.map(arr =>{

val deptid = if(arr(0).nonEmpty) arr(0).toInt else 0

val deptname = arr(1)

val addr = arr(2)

Row(deptid,deptname,addr)

})

val deptDF = sqlContext.createDataFrame(deptRow,schema)

deptDF.registerTempTable("dept")

deptDF.show()

2）DataFrame----->>>> RDD

(1) datarame直接转成rdd然后调用rdd的map算子

(2) dataframe直接调用map算子、flatMap算子

//1.构建DataFrame

val df = sqlContext.read.table("db01.dept")

//2.完成功能需求和转化

//方式一 datarame转成rdd然后调用rdd的map算子

//Row 是dataFrame中数据在RDD的一种表现形式，具备只读属性，只能进行读取操作

val rdd0 = df.rdd

val columns = Array("deptno","deptname","address")

val deptRDD: RDD[(Int, String, String)] = rdd0.map(

row => {

//deptno,deptname,address

val deptno = row.getAs[Int](columns(0))

val deptname = row.getAs[String](columns(1))

val address = row.getAs[String](columns(2))

(deptno,deptname,address)

}

)

// 方式二 dataframe直接调用map算子

val deptRDD1: RDD[(Int, String, String)] = df.map(row =>{

val deptno = row.getAs[Int]("deptno")

val deptname = row.getAs[String]("deptno")

val address = row.getAs[String]("address")

(deptno,deptname,address)

})

7、Dataset

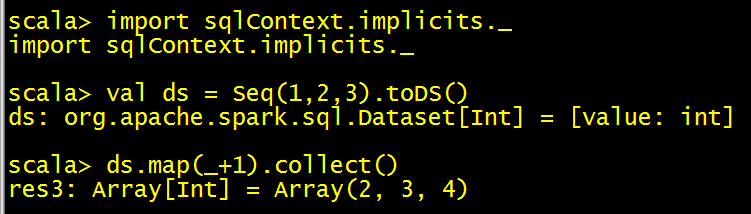
Dataset：一组强类型的对象集合，可以并行地进行转换使用功能或关系操作。

1）内存中的集合数据直接转换成Dataset

要求：引入SparkSQL的隐式转换

Import sqlContext.implicits.\_

val ds = Seq(1,2,3).toDS() ds.map(\_+1).collect()



2）将DataFrame转换为Dataset

要求：DataFrame中的数据类型必须是case class数据类型，并且DataFrame中的数据要求和case class的列名(顺序)一致；DataFrame中的import sqlContext.implicits.\_数据不能存在null

case class Person(name: String, age: Long)

val personDF = sqlContext.read.format("json").

load("file:///opt/cdh-5.3.6/spark/examples/src/main/resources/people.json")

val personDS = personDF.as[Person]

Dataset[Person] = [name: string, age: bigint]

personDS.show

personDF.registerTempTable("tmp01")

val df = sqlContext.sql("select name,age from tmp01 where age is not null")

df.as[Person].show

3）RDD、DataFrame、Dataset之间的关系和区别

(1) 相同点

都是分布式的数据集

DataFrame的底层是RDD

Dataset和DataFrame的底层的执行都是转换成RDD后再执行

1. 区别

① schema 信息

RDD中的数据是没有schema的

DataFrame中的schema是弱类型的，在编译的时候不会进行检查

Dataset中的schema是强类型的，在编译的时候会进行检查

② 序列化机制

RDD和DataFrame底层数据的序列化机制默认是java的序列化机制，可修改为kryo序列化机制

Dataset中使用自定义的数据编码器的形式进行序列化和反序列化

8、SparkSQL的优化

1）hql、sql语句的优化

2）参数优化

(1) spark.sql.autoBroadcastJoinThreshold

指定当表数据小于多少的时候，自动转换成为map join，默认为10M

(2) spark.sql.tungsten.enabled: true，在2.x版本之前最好设置为false

钨丝计划

(3) spark.sql.shuffle.partitions

指定ShuffleMapStage与ShuffleMapStage/ResultStage之间进行shuffle之后，下一个Stage中的分区数(task)的数量; 默认是200

备注：第一个Stage的分区数量不能通过参数控制，因为DataFrame的底层是RDD，所以第一个Stage的分区数量/Task数量，是有读取数据的InputFormat的getSplits方法决定的

spark的默认值是200。Task 过多，会产生很多的任务启动开销，Task多少，每个Task的处理时间过长，容易straggle。